Introducing R

Germán Rodríguez (grodri@princeton.edu)
Princeton University

Spring 2001, last updated Fall 2012
Online version at http://data.princeton.edu/R

Table of Contents

1 Introduction
1.1 The R Language and Environment
1.2 Bibliographic Remarks

2 Getting Started
2.1 The R Console
2.2 Expressions and Assignments
2.3 Vectors and Matrices
2.4 Simple Graphs

3 Reading Data
3.1 Lists and Data Frames
3.2 Free-Format Input
3.3 Fixed-Format Input
3.4 Printing Data and Summaries
3.5 Plotting Data

4 Linear Models
4.1 Fitting a Model
4.2 Examining a Fit
4.3 Extracting Results
4.4 Factors and Covariates
4.5 Regression Splines
4.6 Other Options

5 Generalized Linear Models
5.1 Variance and Link Families
5.2 Logistic Regression
5.3 Updating Models
5.4 Model Selection

6 Conclusion
References

1 Introduction

R is a powerful environment for statistical computing which runs on several platforms. These notes are written especially for users running the Windows version, but most of the material applies to the Mac and Linux versions as well.

1.1 The R Language and Environment

R was first written as a research project by Ross Ihaka and Robert Gentleman, and is now under active development by a group of statisticians called 'the R core team', with a home page at www.r-project.org.

R was designed to be 'not unlike' the S language developed by John Chambers and others at Bell Labs. A commercial version of S with additional features was developed and marketed as S-

Plus by Statistical Sciences, which later become Insightful and is now TIBCO Spotfire. R and S-Plus can best be viewed as two implementations of the S language.

R is available free of charge and is distributed under the terms of the Free Software Foundation's GNU General Public License. You can download the program from the Comprehensive R Archive Network (CRAN). Ready-to-run 'binaries' are available for Windows, Mac OS X, and Linux. The source code is also available for download and can be compiled for other platforms.

These notes are organized in several sections, as shown in the Table of Contents. I have tried to introduce key features of R as they are needed by students in my statistics classes. As a result, I often postpone (or altogether omit) discussion of some of the more powerful features of R as a programming language.

Notes of local interest, such as where to find R at Princeton University, appear in framed boxes and are labeled as such. Permission is hereby given to reproduce these pages freely and host them in your own server if you wish. You may add, edit or delete material in the local notes as long as the rest of the text is left unchanged and due credit is given. Obviously I welcome corrections and suggestions for enhancement.

1.2 Bibliographic Remarks

S was first introduced by Becker and Chambers (1984) in what's known as the 'brown' book. The *new* S language was described by Becker, Chambers and Wilks (1988) in the 'blue' book. Chambers and Hastie (1992) edited a book discussing statistical modeling in S, called the 'white' book. The latest version of the S language is described by Chambers (1998) in the 'green' book, but R is largely an implementation of the versions documented in the blue and white books. Chamber's (2008) latest book focuses on Programming with R.

Venables and Ripley (1994, 1997, 1999, 2002) have written an excellent book on *Modern Applied Statistics with S-PLUS* that is now in its fourth edition. The latest edition is particularly useful to R users because the main text explains differences between S-Plus and R where relevant. A companion volume called *S Programming* appeared in 2000 and applies to both S-Plus and R. These authors have also made available in their website an extensive collection of complements to their books, follow the links at MASS 4.

There is now an extensive and rapidly growing literature on R. Good introductions include the books by Krause and Olson (1987), Dalgaard (2002), and Braun and Murdoch (2007). Beginners will probably benefit from working through the examples in Everitt and Hothorn's (2006) *A Handbook of Statistical Analyses Using R* or Fox's (2002, 2011) *An R Companion to Applied Regression*. Among more specialized books my favorites include Murrell (2005), an essential reference on *R graphics*, Pinheiro and Bates (2000), a book on mixed models, and Therneau and Grambsh's (2000) *Modeling Survival Data*, which includes many analyses using S-Plus as well as

SAS. (Therneau wrote the survival analysis code used in S-Plus and R.) For additional references see the annotated list at R Books.

The official R manuals are available as PDF files that come with the R distribution. These include *An Introduction to R* (a nice 100-page introduction), a manual on *R Data Import/Export* describing facilities for transferring data to and from other packages, and useful notes on *R installation and Administration*. More specialized documents include a draft of the *R Language Definition*, a guide to *Writing R Extensions*, documentation on *R Internals* including coding standards, and finally the massive *R Reference Index* (~3000 pages). The online help facility is excellent. When you install R you get a choice of various help formats. I recommend compiled html help because you get a nice tree-view of the contents, an index, a pretty decent search engine, and nicely formatted help pages. (On Unix you should probably choose html help.)

2 Getting Started

Obviously the first thing you need to do is download a copy of R. The current version is 2.15.1 and was released in June 2012. New releases occur every six months often around April and October. You will find binaries for Windows in the Comprehensive R Archive Network. Find the mirror nearest you and follow the links. The Windows installer is fairly easy to use and, after agreeing to the license terms, lets you choose which components you want to install. Additional packages can always be installed directly from R at a later time. The download website has special notes for users who may have permission issues when installing R in Windows.

Local Notes. R has been installed on OPR's Windows and Unix servers. Just logon to coale or lotka and compute away.

We recommend that you create a shortcut to start R as part of the installation. If you skipped that step just right clicking on the executable `RGui.exe`, drag it to your desktop, and choose 'Create a shotcut here'. It is also a good idea to set R's working directory at this time. Right click on the shortcut, choose 'Properties' and enter your default working directory on the 'Start in' field. The working directory can also be set from within R as explained later. It's a good idea to have a separate folder for each research project.

2.1 The R Console

When R starts you will see a window called the RConsole. This is where you type your commands and see the text results. (Graphs appear in a separate window.) You are prompted to type commands with the greater than symbol. To quit R type

```
> q()
```

Note the parentheses after the q; this is because in R you don't type commands but rather call *functions* to achieve results, even quit! To call a function you type the name followed by the arguments in parentheses. If the function takes no arguments you just type the name followed by left and right parenthesis. (If you forget the parentheses and type just the name of the function, R will list it.)

You should also know about the `help` function, which opens a help window. This function can be called with arguments to obtain help about specific features of R, for example `help(plot)`. A shortcut for help on a topic is a question mark followed by the topic as in `?plot`. If you picked compiled html help during the installation you should get that by default; if not you can always type `options(chmhelp=TRUE)`.

The RConsole allows command editing. You will find that the left and right arrow keys, home, end, backspace, insert, and delete work exactly as you would expect. You also get a command history: the up and down arrow keys can be used to scroll through recent commands. Thus, if you make a mistake all you need to do is press the up key to recall your last command and edit it.

It is possible to prepare commands in a file and then have R execute them using the `source` function. You can send the output to a file instead of the RConsole by using the `sink` function. Both functions take a filename (in quotes) as argument.

Warning: The backslash character has a special meaning to R. When you specify a Windows path in the RConsole you have to: (1) double-up your backslashes, or (2) use forward slashes instead. Thus, if you want to read R code (or data) from a USB drive available as drive E, say, do not type E:\myprogram.r. You have to type E:\\myprogram.r or E:/myprogram.r. (However, when you specify a path in a file open or file save dialog you have to use the native format, with a single backward slash.)

Alternatively, you may prefer to use R interactively and rely on cut and paste to transfer output from the RConsole to a word processing document.

2.2 Expressions and Assignments

R works like a calculator, you type an expression and get the answer:

```
> 1+2
[1] 3
```

The standard arithmetic operators are `+, -, *,` and `/` for add, subtract, multiply and divide, and `^` for exponentiation, so 2^3=8. These operators have the standard precedence, with exponentiation highest and addition/subtraction lowest, but you can always control the order

of evaluation with parentheses. You can use mathematical functions, such as `sqrt`, `exp`, and `log`. For example

```
> log(0.3/(1-0.3))
[1] -0.8472979
```

R also understands the relational operators `<=, <, ==, >, >=` and `!=` for less than or equal, less than, equal, greater than, greater than or equal, and not equal. These can be used to create logical expressions that take values TRUE and FALSE (or T and F for short). Logical expressions may be combined with the logical operators `|` for OR and `&` for AND, as shown further below.

The results of a calculation may be assigned to a named object. The assignment operator in R is `<-`, read as "gets", but by popular demand R now accepts the equal sign as well, so `x <- 2` and `x = 2` both assign the value 2 to a variable (technically an object) named `x`.

Typing a name prints its contents. The name `pi` is used for the constant π. Thus,

```
> s <- pi/sqrt(3)
> s
[1] 1.813799
```

assigns $\pi/\sqrt{3}$ to the variable `s` and then prints the result.

Names may contain letters, numbers or periods, and (starting with 1.9.0) the underscore character, but must start with a letter or period. (I recommend you always start names with a letter.) Thus, `v.one` and `v_one` are valid names but `v one` is not (because it includes a space).

Warning: R is *case sensitive*, `v.one`, `V.one` and `v.One` are all different names.

R objects exist during your session but vanish when you exit. However, you will be asked if you want to save an image of your workspace before you leave. You can also save individual objects to disk, see `help(save)`. (In contrast, S-Plus objects are permanent; they populate--and often overpopulate--your hard disk, staying there until you explicitly remove them.)

Note that assignments are expressions too, you can type `x <- y <- 2` and both x and y will get 2. This works because the assignment `y <- 2` is also an expression that takes the value 2.

Exercise: What's the difference between `x == 2` and `x = 2`? Use the console to find out.

2.3 Vectors and Matrices

So far we have worked with scalars (single numbers) but R is designed to work with **vectors** as well. The function `c`, which is short for catenate (or concatenate if you prefer) can be used to create vectors from scalars or other vectors:

```
> x <- c(1,3,5,7)
> x
[1] 1 3 5 7
```

The colon operator `:` can be used to generate a sequence of numbers

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
```

You can also use the `seq` function to create a sequence given the starting and stopping points and an increment. For example here are eleven values between 0 and 1 in steps of 0.1:

```
> seq(0, 1, 0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Another function that is useful in creating vectors is `rep` for repeat or replicate. For example `rep(3,4)` replicates the number three four times. The first argument can be a vector, so `rep(x,3)` replicates the entire vector x three times. If both arguments are vectors of the *same* size, then each element of the first vector is replicated the number or times indicated by the corresponding element in the second vector. Consider this example:

```
> rep(1:3, 2)
[1] 1 2 3 1 2 3
> rep(1:3, c(2,2,2))
[1] 1 1 2 2 3 3
```

The first call repeats the vector 1:3 twice. The second call repeats each element of 1:3 twice, and could have been written `rep(1:3, rep(2,3))`, a common R idiom.

R operations are *vectorized*. If x is a vector, then `log(x)` is a vector with the logs of the elements of x. Arithmetic and relational operators also work element by element. If x and y are vectors of the same length, then `x + y` is a vector with elements equal to the sum of the corresponding elements of x and y. If y is a scalar it is added to each element of x. If x and y are vectors of different lengths, the shorter one is recycled as needed, perhaps a fractional number of times (in which case you get a warning).

The logical operators `|` for *or* and `&` for *and* also work element by element. (The double operators `||` for *or* and `&&` for *and* work only on the first element of each vector and use shortcut evaluation; they are used mostly in writing R functions and will not be discussed further.)

```
> a = c(TRUE, TRUE, FALSE, FALSE)
> b = c(TRUE, FALSE, TRUE, FALSE)
> a & b
[1]  TRUE FALSE FALSE FALSE
```

The number of elements of a vector is returned by the function `length`. Individual elements are addressed using *subscripts* in square brackets, so `x[1]` is the first element of x, `x[2]` is the second, and `x[length(x)]` is the last.

The subscript can be a vector itself, so `x[1:3]` is a vector consisting of the first three elements of x. A negative subscript excludes the corresponding element, so `x[-1]` returns a vector with all elements of x except the first one.

Interestingly, a subscript can also be a logical expression, in which case you get the elements for which the expression is TRUE. For example to list the elements of x that are less than 5 we use

```
> x[x < 5]
1] 1 2 3 4
```

I read this expression 'x such that x is less than 5'. This works because the subcript `x < 5` is this vector:

```
> x < 5
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

R's (and of course S-Plus's) subscripting facility is extremely powerful. You may find that it takes a while to get used to it, but eventually the language becomes natural.

R also understands matrices and higher dimensional arrays. The following function creates a 3 by 4 matrix and fills it by columns with the numbers 1 to 12:

```
> M = matrix(1:12, 3, 4)
> M
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The elements of a matrix may be addressed using the row and column subscripts in square brackets, separated by a comma. Thus, `M[1,1]` is the first element of `M`.

A subscript may be left blank to select an entire row or column: `M[1,]` is the first row and `M[,1]` is the first column of `M`. Any of the subscripts can be a vector, so `M[1:2,1:2]` is the upper-left 2 by 2 corner of `M`. Try it.

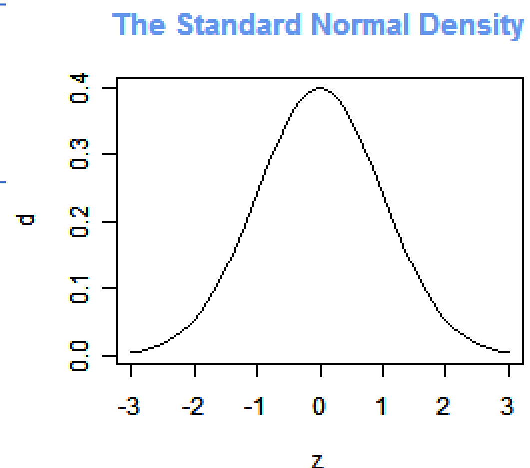
The number of rows and columns of a matrix are returned by the functions `nrow` and `ncol`. To transpose a matrix use the function `t`. The matrix multiplication operator is `%*%`. Matrix inversion is done by the function `solve`. See the linear regression section for an example.

Exercise: How do you list the last element of a matrix?

2.4 Simple Graphs

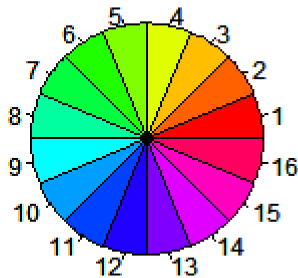
R has very extensive and powerful graphic facilities. In the example below we use `seq` to create equally spaced points between -3 and 3 in steps of 0.1 (that's 61 points). Then we call the function `dnorm` to calculate the standard normal density evaluated at those points, we plot it, and add a title in a nice shade of blue. Note that we are able to add the title to the current plot in a separate call.

```
> z <- seq(-3,3,.1)
> d <- dnorm(z)
> plot(z,d,type="l")
> title("The Standard Normal
Density",col.main="cornflowerblue")
```



Arguments to a function can be specified by position or by name. The plot function expects the first two arguments to be vectors giving the x and y coordinates of the points to be plotted. We also specified the type of plot. Since this is one of many optional parameters (type `?plot` for details), we specified it by name as `type="l"` (the letter el). This indicates that we want the points joined to form a line, rather than the default which is to plot discrete points. Note that R uses the equal sign to specify named arguments to a function.

The title function expects a character string with the title as the first argument. We also specified the optional argument `col.main="cornflowerblue"` to set the color of the title. There are 657 named colors to choose from, type `colors()` to see their names.



The next example is based on a demo included in the R distribution and is simply meant to show off R's use of colors. We use the `pie` function to create a chart with 16 slices. The slices are all the same width, but we fill them with different colors obtained using the `rainbow` function.

```
> pie(rep(1,16),col=rainbow(16))
```

Note the use of the `rep` function to replicate the number one 16 times. To see how one can specify colors and labels for the slices, try calling `pie` with arguments `1:4, c("r", "g", "b","w")` and `col=c("red,"green","blue","white")`.

To save a graph make sure the focus is on the graph window and choose File | Save as, from the menu. You get several choices of format, including postcript, which is good for printing, and windows metafile, which is ideal for embedding your graph in another Windows document.

Most remarkably, you also get the png format, which makes it easy to include R graphs in web pages such as this, particularly now that this format is supported by all major browsers. R also supports jpeg, but I think png is better than jpeg for statistical plots. All graphs on these pages are in png format.

Alternatively, you can copy the graph to the clipboard by choosing File | Copy to clipboard. You get a choice of two formats. I recommend that you use the metafile format because it's more flexible. You can then paste the graph into a word processing or spreadsheet document. You can also print the graph using File | Print.

Exercise: Simulate 20 observations from the regression model $Y = \alpha + \beta x + \epsilon$ using the x vector generated above. Set $\alpha = 1$ and $\beta = 2$. Use standard normal errors generated as `rnorm(20)`, where 20 is the number of observations.

3 Reading and Examining Data

R can handle several types of data, including numbers, character strings, vectors and matrices, as well as more complex data structures. In this section I describe *data frames*, the preferred way to organize data for statistical analysis, explain how to read data from an external file into a data frame, and show how to examine the data using simple descriptive statistics and informative plots.

3.1 Lists and Data Frames

An important data structure that we have not discussed so far is the *list*. A list is a set of objects that are usually named and can be anything: numbers, character strings, matrices or even lists.

Unlike a vector, whose elements must all be of the same type (all numeric, or all character), the elements of a list may have different types. Here's a list with two components created using the function `list`:

```
> person = list(name="Jane", age=24)
```

Typing the name of the list prints all elements. You can extract a component of a list using the extract operator `$`. For example we can list just the name or age of this person:

```
> person$name
[1] "Jane"
> person$age
[1] 24
```

Individual elements of a list can also be accessed using their indices or their names as subscripts. For example we can get the name using `person[1]` or `person["name"]`. (You can use single or double square brackets depending on whether you want a list with the name, which is what we did, or just the name, which would require double brackets as in `person[[1]]` or `person[["name"]]`. The distinction is not important at this point.)

A data frame is essentially a rectangular array containing the values of one or more variables for a set of units. The frame also contains the names of the variables, the names of the observations, and information about the nature of the variables, including whether they are numerical or categorical.

Internally, a data frame is a special kind of list, where each element is a vector of observations on a variable. Data frames look like matrices, but can have columns of different types. This makes them ideally suited for representing datasets, where some variables can be numeric and others can be categorical.

Data frames (like matrices) can also accommodate missing values, which are coded using the special symbol `NA`. Most statistical procedures, however, omit all missing values.

Data frames can be created from vectors, matrices or lists using the function `data.frame`, but more often than not one will read data from an external file, as shown in the next two sections.

3.2 Free-Format Input

Free-format data are text files containing numbers or character strings separated by spaces. Optionally the file may have a header containing variable names. Here's an example of a data file containing information on three variables for 20 countries in Latin America:

	setting	effort	change
Bolivia	46	0	1
Brazil	74	0	10
Chile	89	16	29
Colombia	77	16	25
CostaRica	84	21	29
Cuba	89	15	40
DominicanRep	68	14	21
Ecuador	70	6	0
ElSalvador	60	13	13
Guatemala	55	9	4
Haiti	35	3	0
Honduras	51	7	7
Jamaica	87	23	21
Mexico	83	4	9
Nicaragua	68	0	7
Panama	84	19	22
Paraguay	74	3	6
Peru	73	0	2
TrinidadTobago	84	15	29
Venezuela	91	7	11

This small dataset includes an index of social setting, an index of family planning effort, and the percent decline in the crude birth rate between 1965 and 1975. The data are available on the web at http://data.princeton.edu/wws509/datasets/ in a file called `effort.dat` which includes a header with the variable names.

R can read the data directly from the web:

```
> fpe <- read.table("http://data.princeton.edu/wws509/datasets/effort.dat")
```

The function used to read data frames is called `read.table`. The argument is a character string giving the name of the file containing the data, but here we have given it a fully qualified `url` (uniform resource locator), and that's all it takes.

Alternatively, you could download the data and save them in a local file, or just cut and paste the data from the browser to an editor such as Notepad, and then save them. Make sure the file ends up in R's working directory, which you can find out by typing `getwd()`. If that is not the case you can use a fully qualified path name or change R's working directory by calling `setwd`

with a string argument. Remember to double up your backward slashes (or use forward slashes instead) when specifying paths.

The special symbol `<-`is R's assignment operator, which we have encountered already. Here we assigned the data to an object named `fpe`. To print the data simply type the name of the object.

```
> fpe
              setting effort change
Bolivia            46      0      1
Brazil             74      0     10
    ... output edited ...
Venezuela          91      7     11
```

In this example R detected correctly that the first line in our file was a header with the variable names. It also inferred correctly that the first column had the observation names. (Well, it did so with a little help; I made sure the row names did not have embedded spaces, hence `CostaRica`. Alternatively, I could have used `"Costa Rica"` in quotes as a row name.)

You can always tell R explicitly whether or not you have a header by specifying the optional argument `header=TRUE` or `header=FALSE` to the `read.table` function. This is important if you have a header but lack row names, because R's guess is based on the fact that the header line has one less entry than the next row, as it did in our example.

If your file does not have a header line, R will use the default variable names V1, V2, ..., etc. To override this default use `read.table`'s optional argument `col.names` to assign variable names. This argument takes a vector of names. So, if our file did *not* have a header we could have used the command

```
> fpe = read.table("noheader.dat",
+ col.names=c("setting","effort","change"))
```

Incidentally this is the first time that our command did not fit in a line. R code can be continued automatically in a new line simply by making it obvious that we are not done, for example ending the line with a comma, or having an unclosed left parenthesis. R responds by prompting for more with the continuation symbol `+` instead of the usual prompt `>`.

If your file does not have observation names, R will simply number the observations from 1 to n. You can specify row names using read.table's optional argument `row.names`, which works just like `col.names`; type `?data.frame` for more information.

There are two closely related functions that can be used to get or set variable and observation names at a later time. These are called `names` (for the variable names), and `row.names` (for the

observation names). Thus, if our file did not have a header we could have read the data and then changed the default variable names using the `names` function:

```
> fpe = read.table("noheader.dat")
> names(fpe) = c("setting","effort","change")
```

Technical Note: If you have a background in other programming languages you may be surprised to see a function call on the left hand side of an assignment. These are special 'replacement' functions in R. They extract an element of an object and then replace its value.

In our example all three-variables were numeric. R will handle string variables with no problem. If one of our variables was sex, coded M for males and F for females, R would have created a *factor*, which is basically a categorical variable that takes one of a finite set of values called *levels*. In Section 5 we will use a data frame with categorical variables to illustrate logistic regression. Another way to generate factors is by grouping a numeric covariate. An example appears in Section 4 below.

Exercise: Use a text editor to create a small file with the following three lines:

```
a b c
1 2 3
4 5 6
```

Read this file into R so the variable names are a, b and c. Now delete the first row and read the file again so the variable names are still a, b and c.

3.3 Fixed-Format Input*

Suppose the family planning effort data had been stored in a file containing only the actual data (no country names or variable names) in a fixed format, with social setting in character positions (often called columns) 1-2, family planning effort in positions 3-4 and fertility change in positions 5-6. This is a fairly common way to organize large datasets.

The following call will read the data into a data frame and name the variables:

```
> fpe = read.table("fixedformat.dat",
+  col.names = c("setting", "effort", "change"),
+  sep=c(1, 3, 5))
```

Here I assume that the file in question is called fixedformat.dat. I assign column names just as before, using the `col.names` parameter. The novelty lies in the next argument, called `sep`, which is used to indicate how the variables are separated. The default is white space, which is appropriate when the variables are separated by one or more blanks or tabs. If the data are

separated by commas, a common format with spreadsheets, you can specify `sep=","`. Here we created a vector with the numbers 1, 3 and 5 to specify the character position (or column) where each variable starts. Type `?read.table` for more details.

3.4 Printing Data and Summaries

You can refer to any variable in the `fpe` data frame using the extract operator `$`. For example to look at the values of the fertility change variable, type

```
> fpe$change
```

and R will list a vector with the values of change for the 20 countries. You can also define fpe as your default dataset by "attaching" it to your session:

```
> attach(fpe)
```

If you now type the name `effort` by itself, R will look for it in the `fpe` data frame. If you are done with a data frame you can detach it using `detach(fpe)`.

To obtain simple descriptive statistics on these variables try the `summary` function:

```
> summary(fpe)
    setting          effort          change
 Min.   :35.0   Min.   : 0.00   Min.   : 0.00
 1st Qu.:66.0   1st Qu.: 3.00   1st Qu.: 5.50
 Median :74.0   Median : 8.00   Median :10.50
 Mean   :72.1   Mean   : 9.55   Mean   :14.30
 3rd Qu.:84.0   3rd Qu.:15.25   3rd Qu.:22.75
 Max.   :91.0   Max.   :23.00   Max.   :40.00
```

As you can see, you get the min and max, 1st and 3rd quartiles, median and mean. For categorical variables you get a table of counts. Alternatively, you may ask for a summary of a specific variable. Or use the functions `mean` and `var` for the mean and variance of a variable, or `cor` for the correlation between two variables, as shown below:

```
> mean(effort)
[1] 9.55
> cor(effort,change)
[1] 0.80083
```

Elements of data frames can be addressed using the subscript notation introduced in Section 2.3 for vectors and matrices. For example to list the countries that had a family planning effort score of zero we can use

```
> fpe[effort == 0,]
          setting effort change
Bolivia        46      0      1
Brazil         74      0     10
Nicaragua      68      0      7
Peru           73      0      2
```

This works because the expression `effort == 0` selects the rows (countries) where the effort score is zero, while leaving the column subscript blank selects all columns (variables).

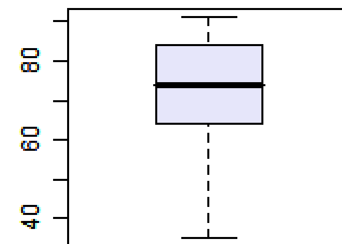
The fact that the rows are named allows yet another way to select elements: by name. Here's how to print the data for Chile:

```
> fpe["Chile",]
      setting effort change
Chile      89     16     29
```

Exercise: Can you list the countries where social setting is high (say above 80) but effort is low (say below 10)? *Hint:* recall the element-by-element logical operator `&`.

3.5 Plotting Data

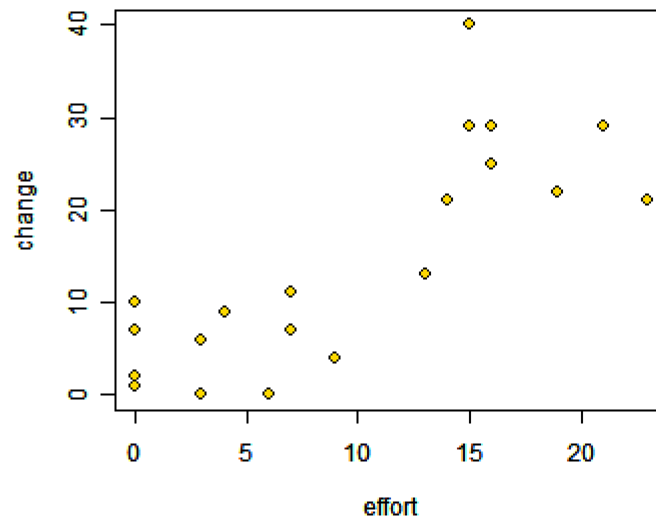
Probably the best way to examine the data is by using graphs. Here's a boxplot of setting. Inspired by a demo included in the R distribution, I used custom colors for the box ("lavender", specifyied using a name R recognizes) and the title (#3366CC, which specifies the red, green and blue components of the color in hexadecimal notation; this particular choice matches the headings on this web page).




```
> boxplot(setting, col="lavender")
> title("Boxplot of Setting", col.main="#3366CC")
```

As noted earlier, R can save a plot as a png or jpeg file, so that it can be included directly on a web page. Other formats available are postscript for printing and windows metafile for embedding in other applications. Note also that you can cut and paste a graph to insert it in another document.

Here's a scatterplot of change by effort, so you can see what a correlation of 0.80 looks like:




```
> plot(effort, change, pch=21, bg="gold")
> title("Scatterplot of Change by Effort", col.main="#3366CC")
```

I used two optional arguments that work well together: `pch=21` selects a special plotting symbol, in this case a circle, that can be colored and filled; and `bf="gold"` selects the fill color for the symbol. I left the perimeter black, but you can change this color with the `col` argument.

To identify points in a scatterplot use the `identify` function. Try the following (assuming the scatterplot is still the active graph):

```
> identify(effort, change, row.names(fpe), ps=9)
```

The first three parameters to this function are the x and y coordinates of the points and the character strings to be used in labeling them. The `ps` optional argument specifies the size of the text in points; here I picked 9-point labels.

Now click within a quarter of an inch of a point and the name of the country should appear in the graph. Which country had the most effort but only moderate change? Which one had the most change?

To quit identifying points right click on the graph and select Stop from the pop-up menu. The function returns the indices of the units selected. (Click on the RConsole to make it the focused window before you type more commands.)

Another interesting plot to try is `pairs`, which draws a scatterplot matrix. In our example try

```
> pairs(fpe)
```

and you will see a 3 by 3 matrix of scatterplots with the variable names down the diagonal and a plot of each variable against every other one.

Before you quit this session consider saving the `fpe` data.frame. To do this use the `save` function

```
> save(fpe, file="fpe.Rdata")
> load("fpe.rdata")
```

The first argument specifies the object to be saved, and the file argument provides the name of a file, which will be in the working directory unless a full path is given. (Remember to double-up your backslashes, or use forward slashes instead.)

By default R saves objet using a compact binary format which is portable across all R platforms. There is an optional argument `ascii` that can be set to TRUE to save the object as ASCII text. This option was handy to transfer R objects across platforms but is no longer needed.

The menu item File | Save Image and its companion File | Load Image can be used to save and load an image of the entire workspace, including all objects that have been created (and not removed) in the session.

Exercise: Use R to create a scatterplot of change by setting, cut and paste the graph into a document in your favorite word processor, and try resizing and printing it. I recommend that you use the windows metafile format for the cut and paste operation.

4 Linear Models

Let us try some linear models, starting with multiple regression and analysis of covariance models, and then moving on to models using regression splines. In this section I will use the data read in Section 3, so make sure the `fpe` data frame is attached to your current session.

4.1 Fitting a Model

To fit an ordinary linear model with fertility change as the response and setting and effort as predictors, try

```
> lmfit = lm( change ~ setting + effort )
```

Note first that lm is a function, and we assign the result to an object that we choose to call lmfit (for linear model fit). This stores the results of the fit for later examination.

The argument to `lm` is a *model formula*, which has the response on the left of the tilde `~` (read "is modeled as") and a Wilkinson-Rogers model specification formula on the right. R uses

- **+** to combine elementary terms, as in A+B
- **:** for interactions, as in A:B;
- ***** for both main effects and interactions, so A*B = A+B+A:B

A nice feature of R is that it lets you create interactions between categorical variables, between categorical and continuous variables, and even between numeric variables (it just creates the cross-product).

4.2 Examining a Fit

Let us look at the results of the fit. One thing you can do with `lmfit`, as you can with any R object, is print it.

```
> lmfit

Call:
lm(formula = change ~ setting + effort)

Coefficients:
(Intercept)      setting       effort
   -14.4511       0.2706       0.9677
```

The output includes the model formula and the coefficients. You can get a bit more detail by requesting a summary:

```
> summary(lmfit)

Call:
lm(formula = change ~ setting + effort)

Residuals:
     Min       1Q   Median       3Q      Max
-10.3475  -3.6426   0.6384   3.2250  15.8530

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -14.4511     7.0938  -2.037 0.057516 .
setting       0.2706     0.1079   2.507 0.022629 *
effort        0.9677     0.2250   4.301 0.000484 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 6.389 on 17 degrees of freedom
Multiple R-Squared: 0.7381,     Adjusted R-squared: 0.7073
F-statistic: 23.96 on 2 and 17 DF,  p-value: 1.132e-05
```

The output includes a more conventional table with parameter estimates and standard errors, as well the residual standard error and multiple R-squared. (By default S-Plus includes the matrix of correlations among parameter estimates, which is often bulky, while R sensibly omits it. If you really need it, add the option `correlation=TRUE` to the call to `summary`.)

To get a hierarchical analysis of variance table corresponding to introducing each of the terms in the model one at a time, in the same order as in the model formula, try the `anova` function:

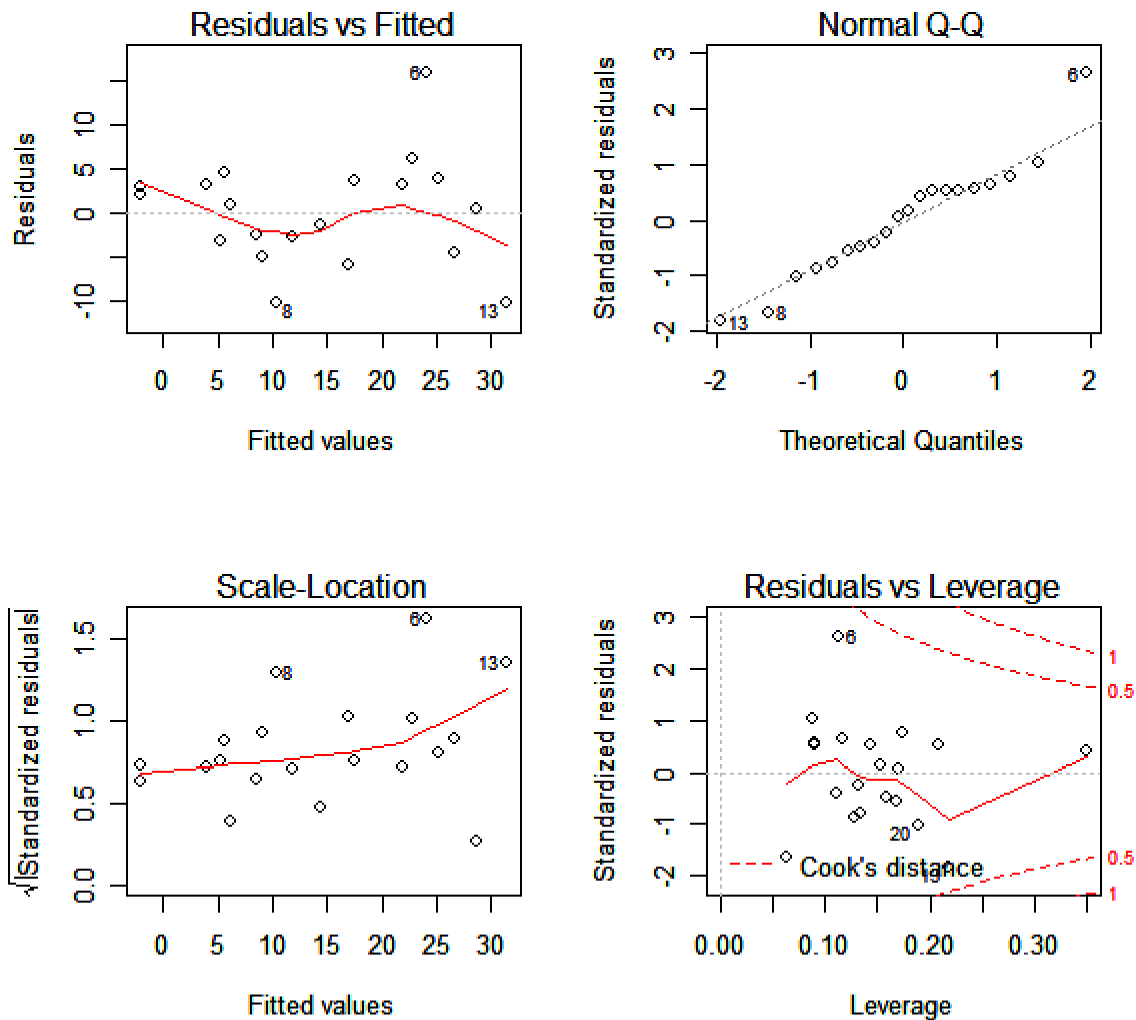
```
> anova(lmfit)
Analysis of Variance Table

Response: change
          Df  Sum Sq Mean Sq F value    Pr(>F)
setting    1 1201.08 1201.08  29.421 4.557e-05 ***
effort     1  755.12  755.12  18.497 0.0004841 ***
Residuals 17  694.01   40.82
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '  1
```

Alternatively, you can plot the results using

```
> plot(lmfit)
```

This will produce a set of four plots: residuals versus fitted values, a Q-Q plot of standardized residuals, a scale-location plot (square roots of standardized residuals versus fitted values, and a plot of residuals versus leverage that adds bands corresponding to Cook's distances of 0.5 and 1.

R will prompt you to click on the graph window or press Enter before showing each plot, but we can do better. Type `par(mfrow=c(2,2))` to set your graphics window to show four plots at once, in a layout with 2 rows and 2 columns. Then redo the graph using `plot(lmfit)`. To go back to a single graph per window use `par(mfrow=c(1,1))`. There are many other ways to customize your graphs by setting high-level parameters, type `?par` to learn more.



Technical Note: You may have noticed that we have used the function `plot` with all kinds of arguments: one or two variables, a data frame, and now a linear model fit. In R jargon plot is a generic function. It checks for the kind of object that you are plotting and then calls the appropriate (more specialized) function to do the work. There are actually many plot functions in R, including `plot.data.frame` and `plot.lm`**.** For most purposes the generic function will do the right thing and you don't need to be concerned about its inner workings.

4.3 Extracting Results

There are some specialized functions that allow you to extract elements from a linear model fit. For example

```
> fitted(lmfit)
```

```
        1         2         3         4         5         6         7
8
-2.004026  5.572452 25.114699 21.867637 28.600325 24.146986 17.496913
10.296380
        ... output edited ...
```

extracts the fitted values. In this case it will also print them, because we did not asign them to anything. (The longer form `fitted.values` is an alias.)

To extract the coefficients use the `coef` function (or the longer form `coefficients`)

```
> coef(lmfit)
(Intercept)     setting      effort
-14.4510978   0.2705885   0.9677137
```

To get the residuals, use the `residuals` function (or the abbreviation `resid`):

```
> residuals(lmfit)
          1           2           3           4           5           6
  3.0040262   4.4275478   3.8853007   3.1323628   0.3996747  15.8530144
        ... output edited ...
```

If you are curious to see exactly what a linear model fit produces, try the function

```
> names(lmfit)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

which lists the named components of a linear fit. All of these objects may be extracted using the `$` operator. However, whenever there is a special extractor function you are encouraged to use it.

4.4 Factors and Covariates

So far our predictors have been continuous variables or *covariates*. We can also use categorical variables or *factors.* Let us group family planning effort into three categories:

```
> effortg = cut(effort, breaks = c(-1, 4, 14, 100),
+    label=c("weak","moderate","strong"))
```

The function `cut` creates a factor or categorical variable. The first argument is an input vector, the second is a vector of breakpoints, and the third is a vector of category labels. Note that

there is one more breakpoint than there are categories. All values greater than the *i*-th breakpoint and less than or equal to the *(i+1)*-st breakpoint go into the *i*-th category. Any values below the first breakpoint or above the last one are coded `NA` (a special R code for missing values). If the labels are omitted, R generates a suitable default of the form "a thru b".

Try fitting the analysis of covariance model:

```
> covfit = lm( change ~ setting + effortg )
> covfit

Call:
lm(formula = change ~ setting + effortg)

Coefficients:
    (Intercept)          setting  effortgmoderate    effortgstrong
        -5.9540           0.1693           4.1439          19.4476
```

As you can see, family planning effort has been treated automatically as a factor, and R has generated the necessary dummy variables for moderate and strong programs treating weak as the reference cell.

Choice of Contrasts: R codes unordered factors using the reference cell or "treatment contrast" method. The reference cell is always the first category which, depending on how the factor was created, is usually the first in alphabetical order. If you don't like this choice, R provides a special function to re-order levels, check out `help(relevel)`.

S codes unordered factors using the *Helmert* contrasts by default, a choice that is useful in designed experiments because it produces orthogonal comparisons, but has baffled many a new user. Both R and S-Plus code ordered factors using polynomials. To change to the reference cell method for unordered factors use the following call

```
> options(contrasts=c("contr.treatment","contr.poly"))
```

Back on to our analysis of covariance fit. You can obtain a hierarchical anova table for the analysis of covariance model using the `anova` function:

```
> anova(covfit)
Analysis of Variance Table

Response: change
          Df  Sum Sq Mean Sq F value    Pr(>F)
setting    1 1201.08 1201.08  36.556 1.698e-05 ***
effortg    2  923.43  461.71  14.053 0.0002999 ***
Residuals 16  525.69   32.86
```

```
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '  1
```

Type `?anova` to learn more about this function.

4.5 Regression Splines

The real power of R begins to shine when you consider some of the other functions you can include in a model formula. First, you can include mathematical functions, for example `log(setting)` is a perfectly legal term in a model formula. You don't have to create a variable representing the log of setting and then use it, R will create it 'on the fly', so you can type

```
> lm( change ~ log(setting) + effort)
```

If you wanted to use orthogonal polynomials of degree 3 on setting, you could include a term of the form `poly(setting,3)`

You can also get R to calculate a well-conditioned basis for regression splines. First you must load the splines library (this step is not needed in S-Plus):

```
> library(splines)
```

This makes available the function `bs` to generate B-splines. For example the call

```
> setting.bs <- bs(setting, knots = c(66,74,84)) + effort )
```

will generate cubic B-splines with interior knots placed at 66, 74 and 84. This basis will use seven degrees of freedom, four corresponding to the constant, linear, quadratic and cubic terms, plus one for each interior knot. Alternatively, you may specify the number of degrees of freedom you are willing to spend on the fit using the parameter `df`. For cubic splines R will choose df-4 interior knots placed at suitable quantiles. You can also control the degree of the spline using the parameter `degree`, the default being cubic.

If you like *natural* cubic splines, you can obtain a well-conditioned basis using the function `ns`, which has exactly the same arguments as `bs` except for degree, which is always three. To fit a natural spline with five degrees of freedom, use the call

```
> setting.ns <- ns(setting, df=5)
```

Natural cubic splines are better behaved than ordinary splines at the extremes of the range. The restrictions mean that you save four degrees of freedom. You will probably want to use two of them to place additional knots at the extremes, but you can still save the other two.

To fit an additive model to fertility change using natural cubic splines on setting and effort with only one interior knot each, placed exactly at the median of each variable, try the following call:

```
> splinefit = lm( change ~ ns(setting, knot=median(setting)) +
+    ns(effort, knot=median(effort)) )
```

Here we used the parameter `knot` to specify where we wanted the knot placed, and the function `median` to calculate the median of setting and effort.

Do you think the linear model was a good fit? Natural cubic splines with exactly one interior knot require the same number of parameters as an ordinary cubic polynomial, but are much better behaved at the extremes.

4.6 Other Options

The `lm` function has several additional parameters that we have not discussed. These include

`data`	to specify a dataset, in case it is not attached
`subset`	to restrict the analysis to a subset of the data
`weights`	to do weighted least squares

and many others; see `help(lm)` for further details. The `args` function lists the arguments used by any function, in case you forget them. Try `args(lm)`.

The fact that R has powerful matrix manipulation routines means that one can do many of these calculations from first principles. The next couple of lines create a model matrix to represent the constant, setting and effort, and then calculate the OLS estimate of the coefficients as $(X'X)^{-1}X'y$:

```
> X <- cbind(1,effort,setting)
> solve( t(X) %*% X ) %*% t(X) %*% change

                [,1]
  [1,] -14.4510978
  [2,]   0.9677137
  [3,]   0.2705885
```

Compare these results **with** `coef(lmfit)`.

5 Generalized Linear Models

Generalized linear models are just as easy to fit in R as ordinary linear model. In fact, they require only an additional parameter to specify the variance and link functions.

5.1 Variance and Link Families

The basic tool for fitting generalized linear models is the `glm` function, which has the folllowing general structure:

```
> glm(formula, family, data, weights, subset, ...)
```

where `...` stands for more esoteric options. The only parameter that we have not encountered before is `family`, which is a simple way of specifying a choice of variance and link functions. There are six choices of family:

Family	Variance	Link
gaussian	gaussian	identity
binomial	binomial	logit, probit or cloglog
poisson	poisson	log, identity or sqrt
Gamma	Gamma	inverse, identity or log
inverse.gaussian	inverse.gaussian	1/mu^2
quasi	user-defined	user-defined

As can be seen, each of the first five choices has an associated variance function (for binomial the binomial variance $\mu(1-\mu)$), and one or more choices of link functions (for binomial the logit, probit or complementary log-log).

As long as you want the default link, all you have to specify is the family name. If you want an alternative link, you must add a `link` argument. For example to do probits you use

```
> glm( formula, family=binomial(link=probit))
```

The last family on the list, `quasi`, is there to allow fitting user-defined models by maximum quasi-likelihood.

5.2 Logistic Regression

We will illustrate fitting logistic regression models using the contraceptive use data shown below:

age	education	wantsMore	notUsing	using
<25	low	yes	53	6
<25	low	no	10	4
<25	high	yes	212	52
<25	high	no	50	10
25-29	low	yes	60	14
25-29	low	no	19	10
25-29	high	yes	155	54
25-29	high	no	65	27
30-39	low	yes	112	33
30-39	low	no	77	46
30-39	high	no	68	78
40-49	low	no	46	48
40-49	high	yes	8	8
40-49	high	no	12	31

The data are available from the datasets section of the website for my generalized linear models course. Visit http://data.princeton.edu/wws509/datasets to read a short description and follow the link to cuse.dat.

Of course the data can be downloaded directly from R:

```
> cuse <- read.table("http://data.princeton.edu/wws509/datasets/cuse.dat",
+     header=TRUE)
> cuse
     age education wantsMore notUsing using
1    <25       low       yes       53     6
2    <25       low        no       10     4
3    <25      high       yes      212    52
4    <25      high        no       50    10
5  25-29       low       yes       60    14
         ... output edited ...
16   40-49      high        no       12    31
```

I specified the `header` parameter as `TRUE`, because otherwise it would not have been obvious that the first line in the file has the variable names. There are no row names specified, so the rows will be numbered from 1 to 16. Print `cuse` to make sure you got the data in alright. Then make it your default dataset:

```
> attach(cuse)
```

Let us first try a simple additive model where contraceptive use depends on age, education and wantsMore:

```
> lrfit <- glm( cbind(using, notUsing) ~
+  age + education + wantsMore , family = binomial)
```

There are a few things to explain here. First, the function is called `glm` and I have assigned its value to an object called `lrfit` (for logistic regression fit). The first argument of the function is a model formula, which defines the response and linear predictor.

With binomial data the response can be either a vector or a matrix with two columns.

- If the response is a vector, it is treated as a binary factor with the first level representing "success" and all others representing "failure". In this case R generates a vector of ones to represent the binomial denominators.
- Alternatively, the response can be a matrix where the first column shows the number of "successes" and the second column shows the number of "failures". In this case R adds the two columns together to produce the correct binomial denominator.

Because the latter approach is clearly the right one for us i used the function **`cbind`** to create a matrix by binding the column vectors containing the numbers using and not using contraception.

Following the special symbol **~** that separates the response from the predictors, we have a standard Wilkinson-Rogers model formula. In this case we are specifying main effects of age, education and wantsMore. Because all three predictors are categorical variables, they are treated automatically as factors, as you can see by inspecting the results:

```
> lrfit

Call:  glm(formula = cbind(using, notUsing) ~ age + education + wantsMore,
      family = binomial)

Coefficients:
 (Intercept)      age25-29      age30-39      age40-49  educationlow
     -0.8082        0.3894        0.9086        1.1892       -0.3250
wantsMoreyes
     -0.8330

Degrees of Freedom: 15 Total (i.e. Null);  10 Residual
Null Deviance:      165.8
Residual Deviance: 29.92        AIC: 113.4
```

Recall that R sorts the levels of a factor in alphabetical order. Because <25 comes before 25-29, 30-39, and 40-49, it has been picked as the reference cell for `age`. Similarly, high is the reference cell for `education` because high comes before low! Finally, R picked no as the base for `wantsMore`.

If you are unhappy about these choices you can (1) use `relevel` to change the base category, or (2) define your own indicator variables. I will use the latter approach by defining indicators for women with high education and women who want no more children:

```
> noMore <- wantsMore == "no"
> hiEduc <- education == "high"
```

Now try the model again:

```
> glm( cbind(using,notUsing) ~ age + hiEduc + noMore, family=binomial)

Call:  glm(formula = cbind(using, notUsing) ~ age + hiEduc + noMore,
     family = binomial)

Coefficients:
(Intercept)     age25-29     age30-39     age40-49       hiEduc       noMore
    -1.9662       0.3894       0.9086       1.1892       0.3250       0.8330

Degrees of Freedom: 15 Total (i.e. Null);  10 Residual
Null Deviance:      165.8
Residual Deviance: 29.92        AIC: 113.4
```

The residual deviance of 29.92 on 10 d.f. is highly significant:

```
> 1-pchisq(29.92,10)
[1] 0.0008828339
```

so we need a better model. One of my favorites introduces an interaction between age and desire for no more children:

```
> lrfit <- glm( cbind(using,notUsing) ~ age * noMore + hiEduc ,
family=binomial)
> lrfit

Call:  glm(formula = cbind(using, notUsing) ~ age * noMore + hiEduc,
    family = binomial)

Coefficients:
    (Intercept)         age25-29         age30-39         age40-49
       -1.80317          0.39460          0.54666          0.57952
         noMore           hiEduc  age25-29:noMore  age30-39:noMore
```

```
        0.06622          0.34065          0.25918          1.11266
age40-49:noMore
        1.36167

Degrees of Freedom: 15 Total (i.e. Null);  7 Residual
Null Deviance:      165.8
Residual Deviance: 12.63        AIC: 102.1
```

Note how R built the interaction terms automatically, and even came up with sensible labels for them. The model's deviance of 12.63 on 7 d.f. is not significant at the conventional five per cent level, so we have no evidence against this model.

To obtain more detailed information about this fit try the `summary` function:

```
> summary(lrfit)

Call:
glm(formula = cbind(using, notUsing) ~ age * noMore + hiEduc,
    family = binomial)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.30027  -0.66163  -0.03286   0.81945   1.73851

Coefficients:
                Estimate Std. Error z value Pr(>|z|)
(Intercept)     -1.80317    0.18018 -10.008  < 2e-16 ***
age25-29         0.39460    0.20145   1.959  0.05013 .
age30-39         0.54666    0.19842   2.755  0.00587 **
age40-49         0.57952    0.34733   1.669  0.09522 .
noMore           0.06622    0.33064   0.200  0.84126
hiEduc           0.34065    0.12576   2.709  0.00676 **
age25-29:noMore  0.25918    0.40970   0.633  0.52699
age30-39:noMore  1.11266    0.37398   2.975  0.00293 **
age40-49:noMore  1.36167    0.48422   2.812  0.00492 **
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '  1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 165.772  on 15  degrees of freedom
Residual deviance:  12.630  on  7  degrees of freedom
AIC: 102.14

Number of Fisher Scoring iterations: 3
```

R follows the popular custom of flagging significant coefficients with one, two or three stars depending on their p-values. Try `plot(lrfit)`. You get the same plots as in a linear model, but

adapted to a generalized linear model; for example the residuals plotted are deviance residuals (the square root of the contribution of an observation to the deviance, with the same sign as the raw residual). The functions that can be used to extract results from the fit include

- `residuals` or `resid`, for the deviance residuals
- `fitted` or `fitted.values`, for the fitted values (estimated probabilities)
- `predict`, for the linear predictor (estimated logits)
- `coef` or `coefficients`, for the coefficients, and
- `deviance`, for the deviance.

Some of these functions have optional arguments; for example, you can extract five different types of residuals, called "deviance", "pearson", "response" (response - fitted value), "working" (the working dependent variable in the IRLS algorithm minus the linear predictor), and "partial" (a matrix of working residuals formed by omitting each term in the model). You specify the one you want using the `type` argument, for example `residuals(lrfit,type="pearson")`.

5.3 Updating Models

If you want to modify a model you may consider using the special function `update`. For example to drop the `age:noMore` interaction in our model one could use

```
> lrfit0 <- update(lrfit, ~ . - age:noMore)
```

The first argument is the result of a fit, and the second an updating formula. The place holder `~` separates the response from the predictors and the dot `.` refers to the right hand side of the original formula, so here we simply remove `age:noMore`. Alternatively, one can give a new formula as the second argument.

The update function can be used to fit the same model to different datasets, using the argument `data` to specify a new data frame. Another useful argument is `subset`, to fit the model to a different subsample. This function works with linear models as well as generalized linear models.

If you plan to fit a sequence of models you will find the `anova` function useful. Given a series of *nested* models, it will calculate the change in deviance between them. Try

```
> anova(lrfit0,lrfit)
Analysis of Deviance Table

Model 1: cbind(using, notUsing) ~ age + noMore + hiEduc
Model 2: cbind(using, notUsing) ~ age + noMore + hiEduc + age:noMore
  Resid. Df Resid. Dev Df Deviance
1        10     29.917
```

```
2          7      12.630  3    17.288
```

Adding the interaction has reduced the deviance by 17.288 at the expense of 3 d.f.

If the argument to anova is a single model, the function will show the change in deviance obtained by adding each of the terms in the order listed in the model formula, just as it did for linear models. Because this requires fitting as many models as there are terms in the formula, the function may take a while to complete its calculations.

The anova function lets you specify an optional test. The usual choices will be "F" for linear models and "Chisq" for generalized linear models. Adding the parameter `test="Chisq"` adds p-values next to the deviances. In our case

```
> anova(lrfit,test="Chisq")

Analysis of Deviance Table
Model: binomial, link: logit
Response: cbind(using, notUsing)
Terms added sequentially (first to last)

           Df Deviance Resid. Df Resid. Dev P(>|Chi|)
NULL                          15    165.772
age         3   79.192        12     86.581 4.575e-17
noMore      1   49.693        11     36.888 1.798e-12
hiEduc      1    6.971        10     29.917     0.008
age:noMore  3   17.288         7     12.630     0.001
```

We can see that all terms were highly significant when they were introduced into the model.

5.4 Model Selection

A very powerful tool in R is a function for stepwise regression that has three remarkable features:

1. It works with generalized linear models, so it will do stepwise logistic regression, or stepwise Poisson regression,
2. It understand about hierarchical models, so it will only consider adding interactions only after including the corresponding main effects in the models, and
3. It understands terms involving more than one degree of freedom, so it it will keep together dummy variables representing the effects of a factor.

The basic idea of the procedure is to start from a given model (which could well be the null model) and take a series of steps by either deleting a term already in the model or adding a

term from a list of candidates for inclusion, called the *scope* of the search and defined, of course, by a model formula.

Selection of terms for deletion or inclusion is based on Akaike's information criterion (AIC). R defines AIC as

-2 maximized log-likelihood + 2 number of parameters

(S-Plus defines it as the deviance minus twice the number of parameters in the model. The two definitions differ by a constant, so differences in AIC are the same in the two environments.) The procedure stops when the AIC criterion cannot be improved.

In R all of this work is done by calling a couple of functions, add1 and drop1, that consider adding or dropping a term from a model. These functions can be very useful in model selection, and both of them accept a **test** argument just like anova.

Consider first drop1. For our logistic regression model,

```
> drop1(lrfit, test="Chisq")
Single term deletions

Model:
cbind(using, notUsing) ~ age + noMore + hiEduc + age:noMore
           Df Deviance     AIC     LRT   Pr(Chi)
          12.630 102.137
hiEduc      1   20.099 107.607   7.469 0.0062755 **
age:noMore  3   29.917 113.425  17.288 0.0006167 ***
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '  1
```

Obviously we can't drop any of these terms. Note that R considered dropping the main effect of education and the age by want no more interaction, but did not examine the main effects of age or want no more, because one would not drop these main effects while retaining the interaction.

The sister function add1 requires a scope to define the additional terms to be considered. In our example we will consider all possible two-factor interactions:

```
> add1(lrfit, ~.^2,test="Chisq")
Single term additions

Model:
cbind(using, notUsing) ~ age + noMore + hiEduc + age:noMore
              Df Deviance     AIC     LRT Pr(Chi)
             12.630 102.137
```

```
age:hiEduc     3    5.798 101.306   6.831 0.07747 .
noMore:hiEduc  1   10.824 102.332   1.806 0.17905
---
Signif. codes:  0  `***'  0.001  `**'  0.01  `*'  0.05  `.'  0.1  ` '  1
```

We see that neither of the missing two-factor interactions is significant by itself at the conventional five percent level. (However, they happen to be jointly significant.) Note that the model with the age by education interaction has a lower AIC than our starting model.

The `step` function will do an automatic search. Here we let it search in a scope defined by all two-factor interactions:

```
> search <- step(additive, ~.^2)
  ... trace output supressed ...
```

The **`step`** function produces detailed trace output that we have supressed. The returned object, however, includes an **`anova`** component that summarizes the search:

```
> search$anova
             Step Df   Deviance Resid. Df Resid. Dev      AIC
1                 NA         NA        10  29.917222 113.4251
2    + age:noMore -3 -17.287669         7  12.629553 102.1375
3    + age:hiEduc -3  -6.831288         4   5.798265 101.3062
4 + hiEduc:noMore -1  -3.356777         3   2.441488  99.9494
```

As you can see, the automated procedure introduced, one by one, all three remaining two-factor interactions, to yield a final AIC of 99.9. This is an example where AIC, by requiring a deviance improvement of only 2 per parameter, may have led to overfitting the data.

Some analysts prefer a higher penalty per parameter. In particular, using log(n) instead of 2 as a multiplier yields BIC, the Bayesian Information Criterion. In our example log(1607) = 7.38, so we would require a deviance reduction of 7.38 per additional parameter. The `step` function accepts **`k`** as an argument, with default 2. You may verify that specifying `k=log(1607)` leads to a much simpler model; not only are no new interactions introduced, but the main effect of education is dropped (even though it is significant).

6 Conclusion

These notes have hardly scratched the surface of R, which has many more statistical functions. These include functions to calculate the density, cdf, and inverse cdf of distributions such as chi-squared, t, F, lognormal, logistic and others. The `survival` library includes methods for the estimation of survival curves, tests of differences between survival curves, and Cox proportional hazards models. The library `nlme` includes code for fitting linear mixed effect models (including

multilevel models) to normally distributed data. Many new statistical procedures are first made available to the research community in the form of S-Plus and R functions.

In addition, R is a full-fledged programming language, with a rich complement of mathematical functions, matrix operations and control structures. If you would like to have a function to compute logits, for example, you can write one just like this:

```
logit <- function(p) {
        log(p/(1-p))
}
```

This function takes as argument a vector of proportions and returns the logits. (The last quantity calculated in a function is returned by default.) Of course this is a very primitive version, because there is no argument checking. A somewhat better version is this:

```
logit <- function(p) {
        if (!is.numeric(p) || any(p1))
                stop("argument must be probabilities between 0 and 1")
        log(p/(1-p))
}
```

The function any called with a logical vector returns true if any element of the vector is true. Of course a value may be in the range (0,1) but so close to either extreme that calculation of the logit could fail; bullet-proofing the function would require more sophisticated code, but the version above is serviceable.

R is an interpreted language but it is reasonably fast, particularly if you take advantage of the fact that operations are vectorized and try to avoid looping. Where efficiency is crucial you can always write a function in a compiled language such as C or Fortran and then call it from R. Some of my work on multilevel generalized linear models uses this approach. To learn more about programming R read Venables and Ripley (2000), Chambers (2008), and the manual on *Writing R Extensions* that comes with the R distribution.

References

Becker, Richard A. and John M. Chambers (1984). *S: An Interactive Environment for Data Analysis and Graphics* Wadsworth, CA.

Becker, Richard A.; John M. Chambers and Allan R. Wilks (1988). *The New S Language*. Chapman & Hall, London

Braun, W. John and Duncan J. Murdoch (2007). *A First Course in Statistical Programming with R*. Cambridge University Press, Cambridge.

Chambers, John M. (1998). *Programming with Data*. Springer, New York.

Chambers, John M (2008). *Software for Data Analysis: Programming with R*. Springer, New York.

Chambers, John M. and Trevor J. Hastie, Editors (1992). *Statistical Models in S*. Chapman & Hall, London.

Dalgaard, Peter (2008). *Introductory Statistics with R*. 2nd Edition Springer, New York.

Everitt, Brian and Torsten Hothorn (2006). *A Handbook of Statistical Analyses Using R*. Chapman & Hall/CRC, Boca Raton, FL.

Fox, John (2002). *An R and S-Plus Companion to Applied Regression*. Sage Publications, Thousand Oaks, CA.

Murrell, Paul (2005). *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL.

Pinheiro, Jose C. and Douglas M. Bates (2000). *Mixed-Effects Models in S and S-Plus*. Springer, New York.

Therneau,Terry M. and Patricia M. Grambsch (2000). *Modeling Survival Data: Extending the Cox Model*. Statistics for Biology and Health. Springer, New York.

Venables, William N. and Brian D. Ripley (2000). *S Programming*. Springer, New York.

Venables, William N. and Brian D. Ripley (2002). *Modern Applied Statistics with S*. Fourth Edition. Springer, New York. (Earlier editions published in 1994, 1997 an 1999.)